7 Parameters and Parameter Tuning

Chapter 3 presented an algorithmic framework that forms the common basis for all evolutionary algorithms. A decision to use an evolutionary algorithm implies that the user adopts the main design decisions behind this framework. Thus, the main algorithm setup follows automatically: the algorithm is based on a population of candidate solutions that is manipulated by selection, recombination, and mutation operators. To obtain a concrete, executable EA, the user only needs to specify a few details. In this chapter we have a closer look at these details, named **parameters**. We discuss the notion of EA parameters and explain why the task of designing an evolutionary algorithm can be seen as the problem of finding appropriate parameter values. Furthermore, we elaborate on the problem of tuning EA parameters and provide an overview of different algorithms that can tune EAs with limited user effort.

7.1 Evolutionary Algorithm Parameters

For detailed discussion of the notion of EA parameters, let us consider a simple GA. As explained in Section 6.1, this is a well-established algorithm with a few degrees of freedom, including the parameters `crossoveroperator`, `crossoverrate`, and `populationsize` (and some others we do not need for the present discussion). To obtain a fully specified, executable version we must provide specific values for these parameters, for instance, setting the parameter `crossoveroperator` to `onepoint`, the parameter `crossoverrate` to 0.5, and the parameter `populationsize` to 100. In principle, we need not distinguish different types of parameters, but intuitively there is a difference between deciding on the crossover operator to be used and choosing a value for the related crossover rate. This difference can be formalized if we distinguish parameters by their domains. The parameter `crossoveroperator` has a finite domain with no sensible distance metric or ordering, e.g., {`onepoint`, `uniform`, `averaging`}, whereas the domain of the parameter $p_c \in [0, 1]$ is a subset of the real numbers $\mathbb{R}$ with the natural structure for real

Table 7.1. Pairs of terms used in the literature to distinguish two types of parameters (variables).

Parameter with an unordered domain	**Parameter with an ordered domain**
qualitative	quantitative
symbolic	numeric
categorical	numerical
structural	behavioral
component	parameter
nominal	ordinal
categorical	ordered

numbers. This difference is essential for searchability. For parameters with a domain that has a distance metric, or is at least partially ordered, one can use heuristic search and optimization methods to find optimal values. For the other type of parameters this is not possible because the domain has no exploitable structure. The only option in this case is sampling.

The difference between two types of parameters has already been noted in evolutionary computing, but various authors use various naming conventions as shown in Table 7.1. Table 7.2 shows an EA-specific illustration with commonly used parameters in both categories. Throughout this book we use the terms **symbolic parameter** and **numeric parameter**. For both types of parameters the elements of the parameter's domain are called **parameter values** and we instantiate a parameter by allocating a value to it.

It is important to note that, depending on particular design choices, one might obtain different numbers of parameters for an EA. For instance, instantiating the symbolic parameter `parentselection` by `tournament` implies a new numeric parameter `tournamentsize`. However, choosing `roulettewheel` does not add any parameters. This example also shows that there can be a hierarchy among parameters. Namely, symbolic parameters may have numeric parameters 'under them'.

7.2 EAs and EA Instances

The distinction between symbolic and numeric parameters naturally supports a distinction between EAs and EA instances. To be specific, we can consider symbolic parameters as high-level ones that define the essence of an evolutionary algorithm, and look at numeric parameters as low-level ones that define a specific variant of this EA. Following this naming convention, an **evolutionary algorithm** is a partially specified algorithm fitting the framework introduced in Chapter 3, where the values to instantiate symbolic parameters are defined, but the numeric parameters are not. Hence, we consider two EAs to be different if they differ in one or more of their symbolic parameters, for instance, if they use different mutation operators. If the values are specified for all parameters, including the numeric ones then we obtain an **evolutionary algorithm instance**. If two EA instances differ only in some of the values of their numeric parameters (e.g., the mutation rate and the tournament size), then we consider them as two variants of the same EA. Table 7.2 illustrates this matter by showing three EA instances belonging to just two EAs.

	A_1	A_2	A_3
symbolic parameters			
`representation`	bitstring	bitstring	real-valued
`recombination`	1-point	1-point	averaging
`mutation`	bit-flip	bit-flip	Gaussian $N(0, \sigma)$
`parentselection`	tournament	tournament	uniform random
`survivorselection`	generational	generational	(μ, λ)
numeric parameters			
p_m	0.01	0.1	0.05
σ	n.a.	n.a.	0.1
p_c	0.5	0.7	0.7
μ	100	100	10
λ	equal μ	equal μ	70
κ	2	4	n.a.

Table 7.2. Three EA instances specified by the symbolic parameters `representation`, `recombination`, `mutation`, `parentselection`, `survivorselection`, and the numeric parameters `mutationrate` (p_m), `mutationstepsize` (σ), `crossoverrate` (p_c), `populationsize` (μ), `offspringsize` (λ), and `tournamentsize` (κ). In our terminology, the instances in columns A_1 and A_2 are just variants of the same EA. The EA instance in column A_3 is an example of a different EA, because it has different symbolic parameter values.

This terminology enables precise formulations and enforces care with phrasing. Observe that the distinction between EAs and EA instances is similar to distinguishing between problems and problem instances. If rigorous terminology is required then the right phrasing is “to apply an EA instance to a problem instance”. However, such rigour is not always needed, and formally inaccurate but understandable phrases like “to apply an EA to a problem” are acceptable if they cannot lead to confusion.

7.3 Designing Evolutionary Algorithms

In the broad sense, algorithm design includes all the decisions needed to specify an algorithm (instance) to solve a given problem (instance). The principal challenge for evolutionary algorithm designers is that the design details, i.e., parameter values, have such a large influence on the performance of the algorithm. Hence, the design of algorithms in general, and EAs in particular, is an optimization problem itself.

To understand this issue, we distinguish three layers: application, algorithm, and design, as shown in Figure 7.1. The whole scheme can be divided into two optimization problems that we refer to as problem solving (the lower part) and algorithm design (the upper part). The lower part consists of an EA instance at the algorithm layer that is trying to find an optimal solution for the given problem instance at the application layer. The upper part contains a design method — the intuition and heuristics of a human user or an automated design strategy — that is trying to find optimal parameter values for the given EA at the algorithm layer. The quality of a given parameter vector is based on the performance of the EA instance using these values.

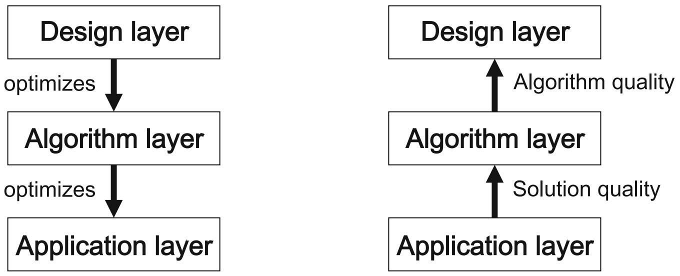

Fig. 7.1. Control flow (left) and information flow (right) through the three layers in the hierarchy of algorithm design. Left: the entity on a given layer optimises the entity on the layer below. Right: the entity on a given layer provides information to the entity on the layer above.

To avoid confusion we use distinct terms to designate the quality function of these optimization problems. In keeping with the usual EC terminology we use the term **fitness** at the application layer, and the term **utility** at the algorithm layer. In the same spirit, we use the term **evaluation** only in relation to fitness, cf. fitness evaluation, and **testing** in relation to utility. With this nomenclature, the problem to be solved by the algorithm designer can be seen as an optimization problem in the space of parameter vectors given some utility function. Solutions of the EA design problem are therefore EA parameter vectors with maximum utility. Table 7.3 provides a quick overview of the resulting vocabulary.

Now we can define the **utility landscape** as an abstract landscape where the locations are the parameter vectors of an EA and the height reflects utility. It is obvious that fitness landscapes, commonly used in EC, have a lot in common with utility landscapes as introduced here. However, despite the obvious analogies, there are some differences we want to note. First of all, fitness values are typically deterministic for most problems. However, utility values are always stochastic, because they reflect the performance of an EA which is a stochastic search method. This implies that the maximum utility needs to be defined in some statistical sense. Consequently, comparing EA parameter vectors can be difficult if the underlying data produced by different EA runs shows a big variance. Second, the notion of fitness is usually strongly related to the objective function of the problem in the application layer, and differences between suitable fitness functions mostly concern arithmetic details. In contrast, the notion of utility depends on the performance metrics used, which reflect the preferences of the user and the context in which the EA is being used. For example, solving a single problem instance just once or repeatedly solving instances of the same problem type represent two very different use cases with different implications for the optimal EA (instance). This issue is dealt with in more detail in Chap. 9.

7.4 The Tuning Problem

To recap, producing an executable EA instance requires specifying values for its parameters. These values determine whether it will find an optimal solution, and whether it will do so efficiently. Parameter tuning is a commonly practised approach to algorithm design, where the values for the parameters are established *before* the run of the algorithm and they remain fixed during the run.

The common way to solve the tuning problem is based on conventions ("mutation rate should be low"), ad hoc choices ("why not use population size 100") and limited experimentation with different values, for example, considering four parameters and five values for each of them. The drawbacks to the first two are obvious. The problems with the experimentation-based approach are traditionally summarised as follows:

- Parameter effects interact (for example, diversity can be created by recombination or mutation), hence they cannot be optimised one by one.
- Trying all different combinations systematically is extremely time consuming. Testing five different values for four parameters leads to $5^4 = 625$ different setups. Performing 100 independent EA runs with each setup implies 62,500 runs with the EA before we can even start the 'real' run.

	Problem solving	**Algorithm design**
Method at work	evolutionary algorithm	design procedure
Search space	solution vectors	parameter vectors
Quality	fitness	utility
Assessment	evaluation	testing

Table 7.3. Vocabulary for problem solving and algorithm design.

- For a numerical parameter, the best parameter values may not even be among the ones we selected for testing. This is because an optimal parameter value could very well lie between the points in the grid we were testing. Increasing the resolution by increasing the number of grid points increases the number of runs exponentially.

This picture becomes even more discouraging if one is after a generally good setup that would perform well on a range of problems. During the history of EAs considerable effort has been spent on finding parameter values that worked well for a number of test problems. A well-known early example is De Jong's thesis [102], which determined recommended values for the probabilities of single-point crossover and bit mutation on what is now called the De Jong test suite of five functions. The contemporary view of EAs, however, acknowledges that specific problem (instances) may require specific EA setups for satisfactory performance [26]. Thus, the scope of 'optimal' parameter settings is necessarily narrow. There are also theoretical arguments that *any* quest for generally good parameter settings is lost a priori, cf. the discussion of the No Free Lunch theorem [467] in Chap. 16.

During the first decades of the evolutionary computing history the tuning issue was largely neglected. Scientific publications did not provide any justification for the parameter values used, nor did they describe the effort spent at arriving to these parameter values. Consequently, it was impossible to tell if the reported EA performance was exceptional (only achievable through intensive tuning) or trivial (obtained by just a bit of tweaking with the parameter values). There was not much research into developing decent tuning procedures either. The idea of optimising GA parameters by a meta-GA was suggested in 1986 [202], but algorithmic approaches to parameter tuning did not receive significant attention for a long time. This situation changed around 2005, when several good tuning algorithms were proposed within a short time interval, such as SPO [42, 41, 43], (iterative) F-race [55, 33], and REVAC [313, 314] and the idea of the meta-GA was also revived [471]. In the last ten years parameter tuning has reached a mature stage: the most important issues are well understood and there are various tuning algorithms available [145, 125]. There are experimental comparisons between some of these methods [378]. However, large-scale adoption by EC researchers and practitioners is still ahead of us.

The basis of the modern approach to parameter tuning is to consider the design of an evolutionary algorithm as a search problem and that of a tuning method as a search algorithm. To this end, it is important that a search algorithm generates a lot of data. In our case, these data concern parameter vectors and their utility values. If one is only interested in an optimal EA configuration then such data are not relevant – finding a good parameter vector is enough. However, these data can be used to reveal information about the given evolutionary algorithm, for instance on its robustness, distribution of solution quality, sensitivity, etc. Thus, adopting the terminology of Hooker

[221], parameter tuning can then be used for competitive as well as scientific testing:

- to configure an EA by choosing parameter values that optimise its performance, and
- to analyse an EA by studying how its performance depends on its parameter values and/or the problems it is applied to.

In both cases, the solutions of a tuning problem depend on the problem(s) to be solved, the EA used, and the utility function that defines how we measure algorithm quality. Adding the tuner to the equation, we obtain Fig. 7.2 which illustrates the generic scheme of parameter tuning in graphical form.

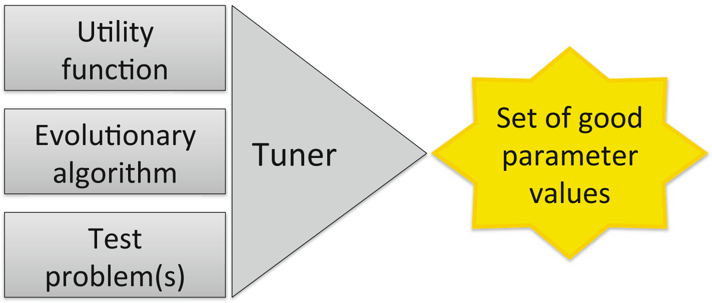

Fig. 7.2. Generic scheme of parameter tuning showing how good parameter values depend on four factors: the problem instance(s) to be solved, the EA used, the utility function, and the tuner itself.

7.5 Algorithm Quality: Performance and Robustness

In general, there are two basic performance measures for EAs, one regarding solution quality and one regarding algorithm speed. Most, if not all, metrics used in EC are based on variations and combinations of these two. Solution quality can be naturally expressed by the fitness function. As for algorithm speed, time or search effort needs to be measured. This can be done by, for instance, the number of fitness evaluations, CPU time, wall-clock time, etc. In [124] and in Chap. 9 of this book we discuss the pros and cons of various time measures. Here we do not go into this issue, we just assume that one of them has been chosen. Then there are three basic combinations of solution quality and computing time that can be used to define algorithm performance in a single run: fix time and measure quality; fix quality and measure time; or fix both and measure completion. For instance:

- Given a maximum running time (computational effort), performance is defined as the best fitness at termination.
- Given a minimum fitness level, performance is defined as the running time (computational effort) needed to reach it.
- Given a maximum running time (computational effort) and a minimum fitness level, performance is defined through the Boolean notion of success: a run is deemed successful if the given fitness is reached within the given time.

Because of the stochastic nature of EAs, a good estimation of performance requires multiple runs on the same problem with the same parameter values and some statistical aggregation of the measures defined for single runs. Doing this for the three measures above gives us the performance metrics commonly used in evolutionary computing:

- MBF (**mean best fitness**),
- AES (**average number of evaluations to a solution**),
- SR (**success rate**).

In Chapter 9 we discuss performance measures in more detail. For this discussion we merely note that it is the choice of performance metrics that determines the utility landscape, and therefore which parameter vector is best. It was recently shown that tuning for different performance measures can yield parameter values that differ in orders of magnitude [376]. This demonstrates why any claim about good parameter values in general, without a reference to the performance measure, should be taken with a pinch of salt.

Regarding **robustness**, the first thing to be noted is that there are different interpretations of this notion in the literature. The existing (informal) definitions do agree that robustness is related to the variance of an algorithm's performance across some dimension, but they differ in what this dimension is. There are indeed more options here, given the fact that the performance of an EA (instance) depends on (1) the problem instance it is solving, (2) the parameter vector it uses, and (3) effects from the random number generator. Therefore, the variance of performance can be considered along three different dimensions: parameter values, problem instances, and random seeds, leading to three different types of robustness.

The first type of robustness is encountered if we are tuning an evolutionary algorithm A on a test suite consisting of many problem instances or test functions. The result of the tuning process is a parameter vector $\bar{p}$ and the corresponding EA instance $A(\bar{p})$ that exhibits good performance over the whole test suite. Note that in this case robustness is defined for EA instances, not EAs. For the historically inclined readers, the famous figures in the classic books of Goldberg [189, page 6], and Michalewicz, [296, page 292], refer to this kind of robustness, with respect to a range of problems.

Another popular interpretation of algorithm robustness is related to performance variations caused by different parameter values. This notion of robustness is defined for EAs. Of course, it is again the EA instance $A(\bar{p})$ whose performance forms the basic measurement, but here we aggregate over parameter vectors. Using such a definition, it is EAs (specified by a particular configuration of the symbolic parameters) that can be compared by their robustness. Finding robust EAs in this sense requires a search through the symbolic parameters.

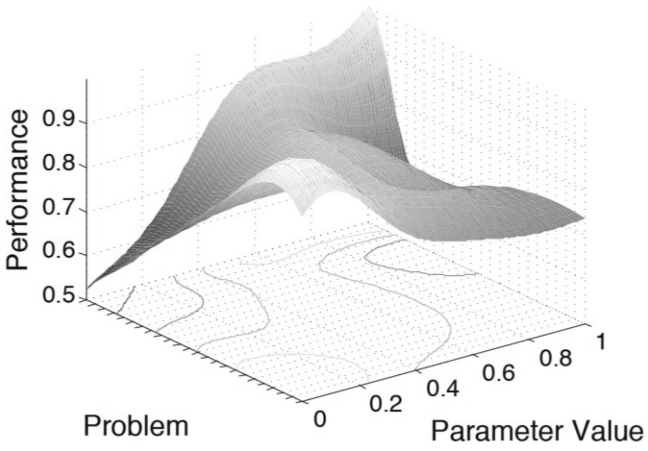

Fig. 7.3. Illustration of the grand utility landscape showing the performance (z) of EA instances belonging to a given parameter vector (x) on a given problem instance (y). Note: The 'cloud' of repeated runs is not shown.

Figure 7.3 shows the grand utility landscape based on all possible combinations of EA parameters and problem instances. For the sake of this illustration we only take a single parameter into account. Thus, we obtain a 3D landscape with one axis, x, representing the values of the parameter and another axis, y, representing the problem instances investigated. (In the general case of n parameters, we have $n+1$ axes here.) The third dimension, z, shows the performance of the EA instance belonging to a given parameter vector on a given problem instance. It should be noted that for stochastic algorithms, such as EAs, this landscape is blurry if the repetitions with different random seeds are also taken into account. That is, rather than one z-value for a pair $\langle x, y \rangle$, we have one z for every run, for repeated runs we get a 'cloud'.

Although this 3D landscape gives the best complete overview of EA performance and robustness, lower-dimensional hyperplanes are also interesting and can be more revealing. The left-hand side of Figure 7.4 shows 2D slices corresponding to specific parameter vectors, i.e., EA instances. Such a slice shows how the performance of an EA instance varies over the range of problem

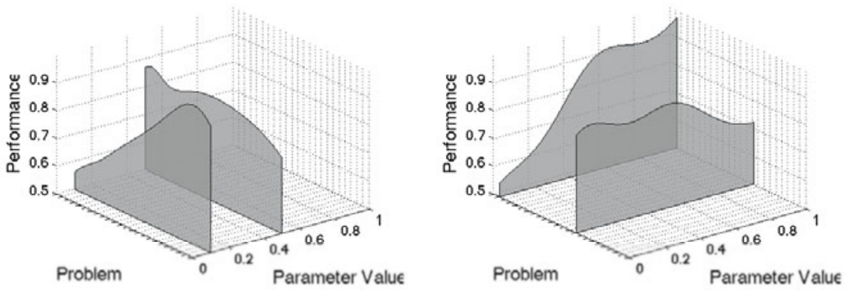

Fig. 7.4. Illustration of parameter-wise slices (left) and problem-wise slices (right) of the grand utility landscape shown in Figure 7.3. (The 'cloud' of repeated runs is not shown.) See the text for explanation and interpretation.

instances. This provides information on robustness with regard to changes in problem specification. Such data are often reported in the literature, often in the form of tables containing the experimental results of one or more EA instances on a predefined test suite, such as the five De Jong functions, the 25 functions of the CEC 2005 contest [420], and the GECCO 2010 test suite [13].

The right-hand side of Figure 7.4 shows 2D slices corresponding to specific problem instances. Each slice shows how the performance of the given EA depends on the parameter values it uses, i.e., its robustness to changes in parameter values. In evolutionary computing such data is hardly ever published. This is a straightforward consequence of the current practice, where parameter values are mostly selected by conventions, ad hoc choices, and very limited experimental comparisons. In other words, usually such data is not even produced, let alone stored and presented. By the increased adoption of tuning algorithms this practice could change, and knowledge about EA parameterization could be collected and disseminated.

7.6 Tuning Methods

In essence, all tuning algorithms work by the GENERATE-and-TEST principle, i.e., by generating parameter vectors and testing them to establish their utility. Considering the GENERATE step, tuners can be then divided into two main categories: *non-iterative* and *iterative* tuners. All non-iterative tuners execute the GENERATE step only once, during initialization, thus creating a fixed set of vectors. Each of those vectors is then tested during the TEST phase to find the best vector in the given set. Hence, one could say that non-iterative tuners follow the INITIALIZE-and-TEST template. Initialization can be done by random sampling, generating a systematic grid in the parameter space, or some space filling set of vectors. Examples of such methods are Latin-Square citemyers2001empirical-model and Taguchi Orthogonal Arrays [424]. Perhaps the best-known method in this category is the frequently used parameter 'optimisation' through a systematic comparison of a few combinations of parameter values, e.g., four mutation rates, four crossover rates, two values for tournament size, and four values for population size. In contrast, iterative tuners do not fix the set of vectors during initialization, but start with a small initial set and create new vectors iteratively during execution. Common examples of such methods are meta-EAs and iterative sampling methods.

Considering the TEST step, we can again distinguish two types of tuners: *single-stage* and *multi-stage procedures*. In both cases the tuners perform a number of tests (i.e., EA runs with the given parameter values) for a reliable estimate of utility. This is necessary because of the stochastic nature of EAs. The difference between the two types is that single-stage procedures perform the same number of tests for each given vector, while multi-stage procedures use a more sophisticated strategy. In general, they augment the TEST step by adding a SELECT step, where only promising vectors are selected for further testing, deliberately ignoring those with a low performance. The best-known method to this end is racing [283].

A further useful distinction between tuning methods can be made by their use of meta-models of the utility landscape. From this perspective tuners can be divided into two major classes: model-free and model-based approaches [226]. Meta-EAs, ParamILS [227], and F-Race [56] belong to the first category. They are 'simply' optimising the given utility landscape, trying to find parameter vectors that maximise the performance of the EA to be tuned. SPO, REVAC, and Bonesa [377] do more than that: during the tuning process they create a model that estimates the performance of an EA for any given parameter vector. In other words, they use meta models or surrogate models [153, 235] that have two advantages. First, meta models reduce the number of expensive utility tests by replacing some of the real tests by model estimates that can be calculated very quickly. Second, they capture information about parameters and their utility for algorithm analysis.

In summary, there exist good tuners that are able to find good parameter values for EAs, and in principle they can tune any heuristic search method with parameters. There are differences between them in terms of efficiency (time needed for tuning), solution quality (performance of the optimised EA), and ease of use (e.g., the number of their own parameters). However, as of 2013, a solid experimental comparison of such tuners is not available. For more details we refer to two recent survey papers [145, 125].

For exercises and recommended reading for this chapter, please visit www.evolutionarycomputation.org.